“All Roads Lead to Rome:” Optimistic Recovery for Distributed Iterative Data Processing

Sebastian Schelter
Stephan Ewen
Kostas Tzoumas
Volker Markl

Technische Universität Berlin, Germany
firstname.lastname@tu-berlin.de

ABSTRACT

Executing data-parallel iterative algorithms on large datasets is crucial for many advanced analytical applications in the fields of data mining and machine learning. Current systems for executing iterative tasks in large clusters typically achieve fault tolerance through rollback recovery. The principle behind this pessimistic approach is to periodically checkpoint the algorithm state. Upon failure, the system restores a consistent state from a previously written checkpoint and resumes execution from that point.

We propose an optimistic recovery mechanism using *algorithmic compensations*. Our method leverages the robust, self-correcting nature of a large class of fixpoint algorithms used in data mining and machine learning, which converge to the correct solution from various intermediate consistent states. In the case of a failure, we apply a user-defined *compensate* function that algorithmically creates such a consistent state, instead of rolling back to a previous checkpointed state. Our optimistic recovery does not checkpoint any state and hence achieves optimal failure-free performance with respect to the overhead necessary for guaranteeing fault tolerance.

We illustrate the applicability of this approach for three wide classes of problems. Furthermore, we show how to implement the proposed optimistic recovery mechanism in a data flow system. Similar to the Combine operator in MapReduce, our proposed functionality is optional and can be applied to increase performance without changing the semantics of programs.

In an experimental evaluation on large datasets, we show that our proposed approach provides optimal failure-free performance. In the absence of failures our optimistic scheme is able to outperform a pessimistic approach by a factor of two to five. In presence of failures, our approach provides fast recovery and outperforms pessimistic approaches in the majority of cases.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel Databases*

Keywords

iterative algorithms; fault-tolerance; optimistic recovery

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505753.

1. INTRODUCTION

In recent years, the cost of acquiring and storing data of unprecedented volume has dropped significantly. As the technologies to process and analyze these datasets are being developed, we face previously unimaginable new possibilities. Businesses can apply advanced data analysis for data-driven decision making and predictive analytics. Scientists can test hypotheses on data several orders of magnitude larger than before, and formulate hypotheses by exploring large datasets.

The analysis of the data is typically conducted using parallel processing platforms in large, shared-nothing commodity clusters. Statistical applications have become very popular, especially in the form of graph mining and machine learning tasks. Many of the algorithms used in these contexts are of *iterative* or *recursive* nature, repeating some computation until a termination condition is met. Their execution imposes serious overhead when carried out with paradigms such as MapReduce [12], where every iteration is scheduled as a separate job and re-scans iteration-invariant data. These shortcomings have led to the proposition of specialized systems [24, 25], as well as to the integration of iterations into data flow systems [8, 14, 26, 27, 31].

The unique properties of iterative tasks open up a set of research questions related to building distributed data processing systems. We focus on improving the handling of machine and network failures during the distributed execution of iterative algorithms. We concentrate on problems where the size of the evolved solution is proportional to the input size and must therefore be partitioned among the participating machines in order to scale to large datasets. The traditional approach to fault tolerance under such circumstances is to periodically persist the application state as checkpoints and, upon failure, restore the state from previously written checkpoints and restart the execution. This pessimistic method is commonly referred to as *rollback recovery* [13].

We propose to exploit the robust nature of many fixpoint algorithms used in data mining to enable an optimistic recovery mechanism. These algorithms will converge from many possible intermediate states. Instead of restoring such a state from a previously written checkpoint and restarting the execution, we propose to apply a user-defined, algorithm-specific *compensation* function. In case of a failure, this function restores a consistent algorithm state and allows the system to continue the execution. Our proposed mechanism eliminates the need to checkpoint intermediate state for such tasks.

We show how to extend the programming model of a parallel data flow system [3] to allow users to specify compensation functions. The compensation function becomes part of the execution plan, and is only executed in case of failures.

In order to show the applicability of our approach to a wide variety of problems, we explore three classes of problems that require distributed, iterative data processing. We start by looking at approaches to carry out link analysis and to compute centralities in large networks. Techniques from this field are extensively used in web mining for search engines and social network analysis. Next, we describe path problems in graphs which include standard problems such as reachability and shortest paths. Last, we look at distributed methods for factorizing large, sparse matrices, a field of high importance for personalization and recommendation mining. For each class, we discuss solving algorithms and provide blueprints for compensation functions. A programmer or library implementor can directly use these functions for algorithms that fall into these classes.

Finally, we evaluate our proposed recovery mechanism by applying several of the discussed algorithms to large datasets. We compare the effort necessary to reach the solution after simulated failures with traditional pessimistic approaches and our proposed optimistic approach. Our results show that our proposed recovery mechanism provides optimal failure-free performance. In case of failures, our recovery mechanism is superior to pessimistic approaches for recovering algorithms that incrementally compute their solution. For non-incremental algorithms that recompute the whole solution in each iteration, we find our optimistic scheme to be superior to pessimistic approaches for recovering from failures in early iterations. This motivates the need for a hybrid approach which we plan to investigate as future work.

1.1 Contributions and Organization

The contributions of this paper are the following:

(1) We propose a novel optimistic recovery mechanism that does not checkpoint any state. Therefore, it provides optimal failure-free performance and simultaneously uses less resources in the cluster than traditional approaches.

(2) We show how to integrate our recovery mechanism into the programming model of a parallel data flow system [3].

(3) We investigate the applicability of our approach to three important classes of problems: link analysis and centrality in networks, path problems in graphs, and matrix factorization. For each class we provide blueprints for generic compensation functions.

(4) We provide empirical evidence that shows that our proposed approach has optimal failure-free performance and fast recovery times in the majority of scenarios.

The rest of the paper is organized as follows. Section 2 positions the proposed optimistic recovery mechanism in relation to existing approaches. Section 3 introduces a parallel programming model for fixpoint algorithms. Sections 4 discusses how to integrate the optimistic recovery for iterations into data flow systems. Section 5 introduces three wide classes of problems for which our proposed recovery mechanism can be applied. Section 6 presents our experimental evaluation. Finally, Sections 7 and 8 discuss related work, conclude, and offer future research directions.

2. A CASE FOR OPTIMISTIC RECOVERY

Realizing fault tolerance in distributed data analysis systems is a complex task. The optimal approach to fault tolerance depends, among other parameters, on the size of the cluster, the characteristics of the hardware, the duration of the data analysis program, the duration of individual stages of the program (i.e., the duration of an iteration in our context), and the progress already made by the program at the time of the failure. Most fault tolerance mechanisms introduce overhead during normal (failure-free) operation, and recovery overhead in the case of failures [13]. We classify recovery mechanisms for large-scale iterative computations in three broad categories, ranging from the most pessimistic to the most optimistic: operator-level pessimistic recovery, iteration-level pessimistic recovery, and the optimistic recovery mechanism proposed in this paper. Pessimistic approaches assume a high probability of failure, whereas optimistic approaches assume low failure probability.

Operator-level recovery, implemented in MapReduce [12], checkpoints the result of every individual program stage (the result of the Map stage in MapReduce). Its sweet spot is very large clusters with high failure rates. This recovery mechanism trades very high failure-free overhead for rapid recovery. In the iterative algorithms setting, such an approach would be desirable if failures occur at every iteration, where such an approach would be the only viable way to allow the computation to finish.

For iterative algorithms, the amount of work per iteration is often much lower than that of a typical MapReduce job, rendering operator-level recovery an overkill. However, the total execution time across all iterations may still be significant. Hence, specialized systems for iterative algorithms typically follow a more optimistic approach. In *iteration-level recovery*, as implemented for example in graph processing systems [24, 25], the result of an iteration as a whole is checkpointed. In the case of failure, all participating machines need to revert to the the last checkpointed state. Iteration-level recovery may skip checkpointing some iterations, trading better failure-free performance for higher recovery overhead in case of a failure. For pessimistic iteration-level recovery to tolerate machine failures, it must replicate the data to checkpoint to several machines, which requires extra network bandwidth and disk space during execution. The overhead incurred to the execution time is immense, in our experiments, we encounter that iterations where a checkpoint is taken take up to 5 times longer than iterations without a checkpoint. Even worse, a pessimistic approach always incurs this overhead, regardless whether a failure happens or not. An extreme point of iteration-level recovery is "no fault tolerance", where checkpoints are never taken, and the whole program is re-executed from scratch in the case of a failure.

On the contrary, our proposed optimistic recovery approach never checkpoints any state. Thus, it provides optimal failure-free performance, as its failure-free overhead is virtually zero (Section 6 experimentally validates this) and at the same time, it requires much less resources in the cluster compared to a pessimistic approach. Furthermore, our optimistic approach only incurs overhead in case of failures, in the form of additional iterations required to compute the solution.

Figure 1 illustrates the optimistic recovery scheme for iterative algorithms: Failure-free execution proceeds as if no fault tolerance is desired. In case of a failure, we finish the current iteration ignoring the failed machines and simultaneously acquire new machines, which initialize the relevant iteration-invariant data partitions. Then, the system applies a user-supplied piece of code, that implements a *compensate* function, on every machine. The compensation function sets the algorithm state to a consistent state from which the algorithm will converge (e.g., if the algorithm computes a probability distribution, the compensation function could have to make sure that all partitions sum to unity). After that, the system proceeds with the execution. The compensation function can be thought of as bringing the computation "back on track", where the errors introduced by the data loss are corrected by the algorithm itself in the subsequent iterations.

In all of our experiments, the optimistic approach shows to be superior, as running additional iterations with a compensation func-

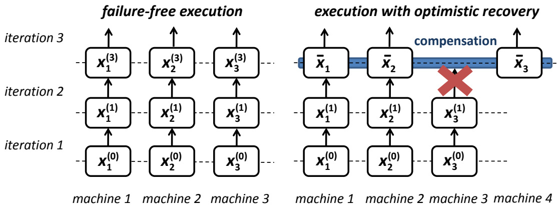

Figure 1: Optimistic recovery with compensations illustrated.

tion results in a shorter overall execution time than writing checkpoints and repeating iterations with a pessimistic approach. Therefore, we conjecture that our proposed approach is the best fit for running iterative algorithms on clusters with moderate failure rates, given that a compensation for the algorithm to execute is known. Note that for many algorithms, it is easy to write a compensation function whose application will provably lead the algorithm to convergence (cf., Section 5, where we provide those for a variety of algorithms). Our experiments in Section 6 show that our proposed optimistic recovery combines optimal failure-free performance with fast recovery and at the same time outperforms a pessimistic approach in the vast majority of cases.

3. PRELIMINARIES

3.1 Fixpoint Algorithms

We restrict our discussion to algorithms that can be expressed by a general fixpoint paradigm taken from Bertsekas and Tsitsiklis [4]. Given an n-dimensional state vector $x \in \mathbb{R}^n$, and an update function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^n$, each iteration consists of an application of f to x: $x^{(t+1)} = f(x^{(t)})$. The algorithm terminates when we find the fixpoint x^* of the series $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, ..., such that $x^* = f(x^*)$.

The update function f is decomposed into component-wise update functions f_i. The function $f_i(x)$ updates component $x_i^{(t)}$ such that $x_i^{(t+1)} = f_i(x_1^{(t)}, x_2^{(t)}, \dots, x_n^{(t)})$ for $i = 1, \dots, n$.

An update function f_i might only depend on a few components of the state $x^{(t)}$. The structure of these *computational dependencies of the data* is described by the *dependency graph* $G_{dep} = (N, E)$, where the vertices $N = \{x_1, \dots, x_n\}$ represent the components of x, and the edges E represent the dependencies among the components: $(i, j) \in E \Leftrightarrow f_i$ depends on x_j.

The dependencies between the components might be subject to additional parameters, e.g., distance, conditional probability, transition probability, etc, depending on the semantics of the application. We denote the parameter of the dependency between components x_i and x_j with a_{ij}. This dependency parameter can be thought of as the weight of the edge e_{ij} in G_{dep} . We denote the adjacent neighbors of x_i in G_{dep}, i.e., the computational dependencies of x_i, as Γ_i (cf. Figure 2).

3.2 Programming Model

We define a simple programming model for implementing iterative algorithms based on the introduced notion of fixpoints. Each algorithm is expressed by two functions. The first function is called *initialize* and creates the components of the initial state $x^{(0)}$:

$$\text{initialize} : i \rightarrow x_i^{(0)}$$

The second function, termed *update*, implements the component update function f_i. This function needs as input the states of the components which x_i depends on, and possibly parameters for these dependencies. The states and dependency parameters necessary for recomputing component x_i at iteration t are captured in the *dependency set* $D_i^{(t)} = \{(x_j^{(t)}, a_{ij}) \,|\, x_j^{(t)} \in \Gamma_i\}$. The function computes $x_i^{(t+1)}$ from the dependency set $D_i^{(t)}$:

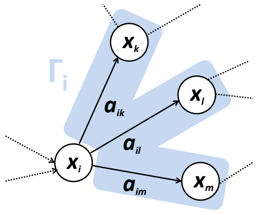

Figure 2: Adjacent neighbors Γ_i of x_i in the dependency graph, a representation of x_i's computational dependencies.

$$\text{update} : D_i^{(t)} \rightarrow x_i^{(t+1)}$$

We refer to the union $D^{(t)}$ of the dependency sets $D_i^{(t)}$ for all components i as the *workset*.

In order to detect the convergence of the fixpoint algorithm in distributed settings, we need two more functions. The function aggregate : $(x_i^{(t+1)}, x_i^{(t)}, agg) \rightarrow agg$ incrementally computes a global aggregate agg from the current value $x_i^{(t+1)}$ and the previous value $x_i^{(t)}$ of each component x_i. It is a commutative and associative function. The function converged : $agg \rightarrow \{true, false\}$ decides whether the algorithm has converged by inspecting the current global aggregate.

Example: PageRank [29] is an iterative method for ranking web pages based on the underlying link structure. Initially, every page has the same rank. At each iteration, every page uniformly distributes its rank to all pages that it links to, and recomputes its rank by adding up the partial ranks it receives. The algorithm converges when the ranks of the individual pages do not change anymore.

For PageRank, we start with a uniform rank distribution by initializing each x_i to $\frac{1}{n}$, where n is the total number of vertices in the graph. The components of x are the vertices in the graph, each vertex depends on its incident neighbors, and the dependency parameters correspond to the transition probabilities between vertices. At each iteration, every vertex recomputes its rank from its incident neighbors proportionally to the transition probabilities:

$$\text{update} : D_i^{(t)} \rightarrow 0.85 \sum_{D_i^{(t)}} a_{ij} x_j^{(t)} + 0.15 \, \frac{1}{n}.$$

The aggregation function computes the L1-norm of the difference between the previous and the current PageRank solution, by summing up the differences between the previous and current ranks, and the algorithm converges when this difference becomes less than a given threshold.

3.3 Parallel Execution

This fixpoint mathematical model is amenable to a simple parallelization scheme. Computation of $x_i^{(t+1)}$ involves two steps:

1. Collect the states $x_j^{(t)}$ and parameters a_{ij} for all dependencies $j \in \Gamma_i$.
2. Form the dependency set $D_i^{(t)}$, and invoke update to obtain $x_i^{(t+1)}$.

Assume that the vertices of the dependency graph are represented as tuples $(n, x_n^{(t)})$ of component index n and state $x_n^{(t)}$. The edges of the dependency graph are represented as tuples (i, j, a_{ij}), indicating that component x_i depends on component x_j with parameter a_{ij}. If the datasets containing the states and dependency graph are co-partitioned, then the first step can be executed by a local join between vertices and their corresponding edges on the component index $n = j$. For executing step 2, the result of the join is projected to the tuple $(i, x_j^{(t)}, a_{ij})$ and grouped on the component index i to form the dependency set $D_i^{(t)}$. The $\mathtt{update}$ function then aggregates $D_i^{(t)}$ to compute the new state $x_i^{(t+1)}$.

This parallelization scheme, which can be summarized by treating a single iteration as *a join followed by an aggregation*, is a special case of the *Bulk Synchronous Parallel* (BSP) [32] paradigm. BSP models parallel computation as local computation (the *join* part) followed by message passing between independent processors (the *aggregation* part). Analogously to the execution of a *superstep* in BSP, we assume that the execution of a single iteration is synchronized among all participating computational units.

4. INTEGRATING COMPENSATIONS

4.1 Fixpoint Algorithms as Data Flows

We illustrate and prototype our proposed recovery scheme using a general data flow system with a programming model that extends MapReduce. For implementation, we use Stratosphere [3], a massively parallel data analysis system which offers dedicated support for iterations, a functionality necessary for efficiently running fixpoint algorithms [14]. In the following, we will use the operator notation from Stratosphere. However, the ideas presented in this paper are applicable to other data flow systems with support for iterative or recursive queries.

To ensure efficient execution of fixpoint algorithms, Stratosphere offers two distinct programming abstractions for iterations [14]. In the first form of iterations, called *bulk iterations*, each iteration completely recomputes the state vector $x^{(t+1)}$ from the previous iteration's result $x^{(t)}$.

Figure 3 shows a generic logical plan for modeling fixpoint algorithms as presented in Section 3.2 using the bulk iteration abstraction (ignore the dotted box for now). The input consists of records $(n, x_n^{(t)})$ representing the components of the state $x^{(t)}$ on the one hand, and of records (i, j, a_{ij}) representing the dependencies with parameters on the other hand (cf. Section 3.3). The data flow program starts with a "dependency join", performed by a *Match*[1] operator, which joins the components $x_i^{(t)}$ with their corresponding dependencies and parameters to form the elements $(i, x_j^{(t)}, a_{ij})$ of the dependency set $D_i^{(t)}$. The "update aggregation" operator groups the result of the join on the component index i to form the dependency set $D_i^{(t)}$, and applies the $\mathtt{update}$ function to compute $x_i^{(t+1)}$ from the dependency set. The "convergence check" operator joins and compares the newly computed state $x_i^{(t+1)}$ with the previous state $x_n^{(t)}$ on $n = i$. From the difference between the components, the operator computes a global aggregate using the $\mathtt{aggregate}$ function. A mechanism for efficient computation of distributive aggregates, similar to the aggregators in Pregel [25], invokes the $\mathtt{converged}$ function to decide whether to trigger a successive iteration (for simplicity reasons we omit this from the figure). If the algorithm has not converged, the "convergence check" operator feeds the records $(i, x_i^{(t+1)})$ that form $x^{(t+1)}$ into the next iteration.

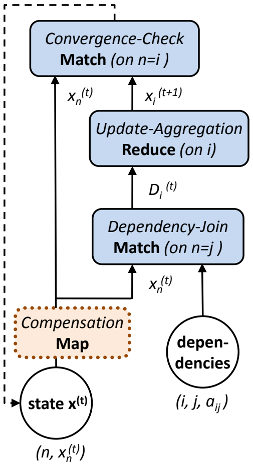

Figure 3: *Bulk* fixpoint algorithm in Stratosphere.

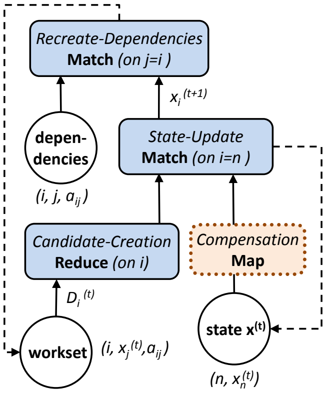

Figure 4: *Incremental* fixpoint algorithm in Stratosphere.

The second form of iterations, called *incremental iterations* [14], are optimized for algorithms that only partially recompute the state $x^{(t)}$ in each iteration. A generic logical plan for fixpoint algorithms using this strategy is shown in Figure 4. Like the bulk iteration variant, the plan models the two steps from the fixpoint model described in Section 3.3. It differs from the bulk iteration plan in that it does not feed back the entire state at the end of an iteration, but only the dependency sets $D_i^{(t)}$ for the fraction of components that will be updated in the next iteration (c.f., the feedback edge from the "recreate dependencies" operator to the workset in Figure 4). The system updates the state of the algorithm using the changed components rather than fully recomputing it. Hence, this plan exploits the fact that certain components converge earlier than others, as dependency sets are fed back selectively only for those components whose state needs to be updated. In addition to the two inputs from the bulk iteration variant, this plan has a third input with the initial version of the workset $D^{(0)}$. The creation of $D^{(0)}$ depends on the semantics of the application, but in most cases, $D^{(0)}$ is simply the union of all initial dependency sets.

In Figure 4, the "candidate creation" operator groups the elements of the workset on the component index i to form the dependency sets $D_i^{(t)}$ and applies the $\mathtt{update}$ function to compute a candidate update for each $x_i^{(t+1)}$ from the corresponding dependency set. The "state update" operator joins the candidate with the corresponding component from $x^{(t)}$ on the component index i and decides whether to set $x_i^{(t+1)}$ to the candidate value. If an update occurs, the system emits a record $(i, x_i^{(t+1)})$ containing the updated component to the "recreate dependencies" operator, which joins the updated components with the dependencies and parameters. In addition, the records are efficiently merged with the current state (e.g., via index merging - see rightmost feedback edge in the figure). As in the bulk iteration variant, we represent the dependencies as (i, j, a_{ij}), and join them on $j = i$. The operator emits elements of the dependency sets to form the workset $D^{(t+1)}$ for the next iteration. The algorithm converges when an iteration creates no new dependency sets, i.e. when the workset $D^{(t+1)}$ is empty. We restrict ourselves to algorithms that follow the plans of Figures 3 and 4, where the dataset holding the dependencies can be

[1] A *Match* is a second-order function performing an equi-join followed by a user-defined first-order function applied to the join result [3].

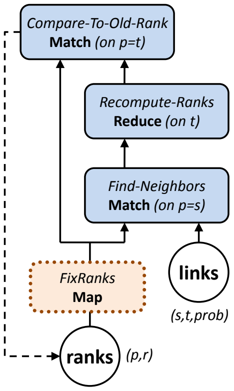

Figure 5: PageRank as iterative data flow.

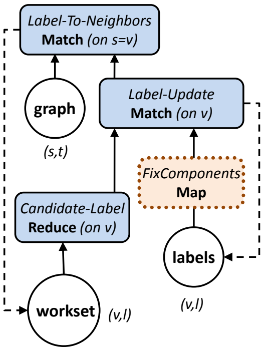

Figure 6: Connected Components as iterative data flow.

either a materialized or a non-materialized view, i.e., it may be the result of an arbitrary plan.

4.2 Recovering Bulk Iterations

To integrate optimistic recovery to the bulk iterations model, we introduce a "compensate" operator that takes as input the current state of components $x^{(t)}$ (dotted box in Figure 3). The output of the compensation operator is input to the dependency join operator. The system only executes the compensate operator after a failure: e.g., in case of a failure at iteration t, the system finishes the current iteration and activates the additional operator in the plan for iteration $t+1$.

We note that the compensate operator can be an arbitrary plan in itself. However, we found that compensation functions embedded in a simple Map operator are adequate for a wide class of algorithms (see Section 5). In addition, using a compensation function embedded in a Map operator ensures fast recovery times without the need for data shuffling. The compensation function often leverages lightweight meta-data from previous iterations that are stored in a distributed fault-tolerant manner using a distributed locking service [9].

PageRank: Figure 5 shows the data flow plan, derived from the general plan in Section 4.1, for the PageRank algorithm.

The initial state $x^{(0)}$ consists of all pages p and initial ranks r. The dependencies $(s, t, prob)$ consist of the edges $s \rightarrow t$ of the graph, weighted by the transition probability $prob$ from page s to page t. The "find neighbors" operator joins the pages with their outgoing links on $p = s$ and creates tuples (t, c) holding the partial rank $c = r * prob$ for the neighbors. The "recompute ranks" operator groups the result of the "find neighbors" join on the target page t to recompute its rank using PageRank's update formula: $0.85*\sum c+0.15/n$. It emits a tuple (t, r_{new}) that contains the new rank to the "compare to old rank" operator, which joins these tuples with the previous ranks on $p = t$ and initiates the distributed aggregation necessary for the convergence check. Finally, the "compare to old rank" operator emits tuples (p, r) containing the pages and recomputed ranks. If PageRank has not converged, these tuples form the input to the next iteration $t + 1$.

In case of a failure, the system activates the additional Map operator called "fix ranks" in the plan, as shown in Figure 5. This operator executes the compensation function. A simple compensation approach is to re-initialize the ranks of vertices in failed partitions uniformly and re-scale the ranks of the non-failed vertices, so that all ranks still sum to unity (cf. Algorithm 1). A requirement for this mechanism is that the system knows the total number of vertices n, and keeps an aggregate statistic about of the current number of vertices $n_{\text{nonfailed}}$ and the current total rank $r_{\text{nonfailed}}$. Note that these statistics can be maintained at virtually zero cost when computed together with the convergence check by the distributed aggregation mechanism.

Algorithm 1 Compensation function for PageRank.

```
function FIX-RANKS-UNIFORM((pid, $r$), ($n$, $n_{\text{nonfailed}}$, $r_{\text{nonfailed}}$))
    if pid is in failed partition then return (pid, $1/n$)
    else return (pid, $(n_{\text{nonfailed}} \cdot r)/(n \cdot r_{\text{nonfailed}})$)
```

4.3 Recovering Incremental Iterations

Analogously to bulk iterations, we compensate a large class of incremental iterations by a simple Map operation, applied to the state at the beginning of the iteration subsequent to a failure. Additionally, for incremental iterations, the system needs to recreate all dependency sets required to recompute the lost components. This is necessary because the recomputation of a failed component might depend on another already converged component whose state is not part of the workset anymore. In the plan from Figure 4, the "recreate dependencies" operator produces dependency sets from all components of the state that were updated in the failing iteration, including components that were recreated or adjusted by the compensation function. The system hence only needs to recreate the necessary dependency sets originating from components that were not updated in the failing iteration. To identify those non-updated components, the "state-update" operator internally maintains a timestamp (e.g., the iteration number) for each component, indicating its last update. In the iteration subsequent to a failure, a record for each such component is emitted by the "state update" operator to the "recreate dependencies" operator. The "recreate dependencies" operator joins these record with the dependencies, creating the dependency sets $D^{(t+1)}$ for all components depending on it. From the output of the "recreate dependencies" operator, we prune all elements that do not belong to a lost component from these extra dependency sets[2]. By means of this optimization, the system does not unnecessarily recompute components that did not change in their dependent components and are not required for recomputing a failed component.

Connected Components [21]: This algorithm identifies the connected components of an undirected graph, the maximum cardinality sets of vertices that can reach each other. We initially assign to each vertex v a unique numeric label which serves as the vertex's state x_i. At every iteration of the algorithm, each vertex replaces its label with the minimum label of its neighbors. In our fixpoint programming model, we express it by the function $\text{update} : D_i^{(t)} \rightarrow \min_{D_i^{(t)}} (x_j^{(t)})$. At convergence, the states of all vertices in a connected component are the same label, the minimum of the labels initially assigned to the vertices of this component.

Figure 6 shows a data flow plan for Connected Components, derived from the general plan for incremental iterations discussed in Section 4.1. There are three inputs. The initial state $x^{(0)}$ consists of the initial labels, a set of tuples (v, l) where v is a vertex of the graph and l is its label. Initially, each vertex has a unique label. The dependencies and parameters for this problem map directly to the graph structure, as each vertex depends on all its neighbors. We

[2] This is easily achieved by evaluating the partitioning function on the element's join key and checking whether it was assigned to a failed machine.

represent each edge of the graph by a tuple (s, t), referring to the source vertex s and target vertex t of the edge. The initial workset $D^{(0)}$ consists of candidate labels for all vertices, represented as tuples (v, l), where l is a label of v's neighbor (there is one tuple per neighbor of v).

Algorithm 2 Compensation function for Connected Components.

```
function FIX-COMPONENTS($(v, c)$)
    if $v$ is in failed partition then return $(v, v)$
    else return $(v, c)$
```

First, the "candidate label" operator groups the labels from the workset on the vertex v and computes the minimum label l_{new} for each vertex v. The "label update" operator joins the record containing the vertex and its candidate label l_{new} with the existing entry and its label on v. If the candidate label l_{new} is smaller than the current label l, the system updates the state $x_v^{(t)}$ and the "label update" operator emits a tuple (v, l_{new}) representing the vertex v with its new label l_{new}. The emitted tuples are joined with the graph structure on $s = v$ by the "label to neighbors" operator. This operator emits tuples (t, l), where l is the new label and t is a neighbor vertex of the updated vertex v. These tuples form the dependency sets for the next iteration. As $D^{(t+1)}$ only contains vertices for which a neighbor updated its label, we do not unnecessarily recompute components of vertices without a change in their neighborhood in the next iteration. The algorithm converges once the system observes no more label changes during an iteration by observing the number of records emitted from the "label update" operator.

As discussed, the system automatically recomputes the necessary dependency sets in case of a failure in iteration t. Therefore, analogously to bulk iterations, the programmer's only task is to provide a record-at-a-time operation which applies the compensation function to the state $x^{(t+1)}$. Here, it is sufficient to set the label of vertices in failed partitions back to its initial value (Algorithm 2).

5. COMPENSABLE PROBLEMS

In order to demonstrate the applicability of our proposed approach to a wide range of problems, we discuss three classes of large-scale data mining problems which are solved by fixpoint algorithms. For each class, we list several problem instances together with a brief mathematical description and provide a generic compensation function as blueprint.

5.1 Link Analysis and Centrality in Networks

We first discuss problems that compute and interpret the *dominant eigenvector of a large, sparse matrix* representing a network. A well known example of such a problem is *PageRank* [29], which computes the relevance of web pages based on the underlying link structure of the web. PageRank models the web as a Markov chain and computes the chain's steady-state probabilities. This reduces to finding the dominant eigenvector of the transition matrix representing the Markov chain. Related methods of ranking vertices are *eigenvector centrality* [6] and *Katz centrality* [22] which interpret the dominant eigenvector (or a slight modification of it) of the adjacency matrix of a network. Another example of an eigenvector problem is *Random Walk With Restart* [21], which uses a random walk biased towards a source vertex to compute the proximity of the remaining vertices of the network to this source vertex. *LineRank* [20] was recently proposed as a scalable substitute for betweeness centrality, a flow-based measure of centrality in networks. Similarly to the previously mentioned techniques, it relies on computing the dominant eigenvector of a matrix, in this case the transition matrix induced by the line graph of the network. The dominant *eigenvector of the modularity matrix* [28] can be used to split the network into two communities of vertices with a higher than expected number of edges between them.

Solutions for the problems of this class are usually computed by some variant of the *Power Method* [18], an iterative algorithm for computing the dominant eigenvector of a matrix M. An iteration of the algorithm consists of a matrix-vector multiplication followed by normalization to unit length. To model the Power Method as a fixpoint algorithm, we operate on a sparse $n \times n$ matrix, whose entries correspond to the dependency parameters. We uniformly initialize each component of the estimate of the dominant eigenvector to $\frac{1}{\sqrt{n}}$. The update function computes the dot product of the i-th row of the matrix M and the previous state $x^{(t)}$. We express it by the function $\text{update} : D_i^{(t)} \to \frac{1}{\|x^{(t)}\|_2} \sum_{D_i^{(t)}} a_{ij} x_j^{(t)}$ in our fixpoint programming model. The result is normalized by the L2-norm of the previous state, which can be efficiently computed using a distributed aggregation. The algorithm converges when the L2-norm of the difference between the previous and the current state becomes less than a given threshold.

For failure compensation, it is enough to uniformly re-initialize the lost components to $\frac{1}{\sqrt{n}}$, as the power method will still provably converge to the dominant eigenvector afterwards [18]:

$$\text{compensate} : x_i^{(t)} \to \begin{cases} \frac{1}{\sqrt{n}} & \text{if } i \text{ belongs to a failed partition} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$

This function suffices as base for compensating all of the above problems when they are solved by the power method.

5.2 Path Enumeration Problems in Graphs

The second class of problems we discuss can be seen as variants of enumerating paths in large graphs and aggregating their weights. Instances of this problem class include *single-source reachability*, *single-source shortest paths*, *single-source maximum reliability paths*, *minimum spanning tree* [19], as well as finding the *connected components* [21] of an undirected graph.

Instances of this problem class can be solved by the *Generalized Jacobi algorithm* [19], which is defined on an idempotent semiring $(S, \oplus, \otimes)$. The state vector $x^{(0)}$ is initialized using the identity element e of $\otimes$ for the source vertex identity element ϵ of $\oplus$ for all other vertices. The algorithm operates on a given, application-specific graph. The dependency parameters correspond to the edge weights of this graph, taken from the set S, on which the semiring is defined. At iteration t, the algorithm enumerates paths of length t with relaxation operations, computes the weight of the paths with the $\otimes$ operation and aggregates these weights with the second, idempotent operation $\oplus$. In our fixpoint programming model, we express it by the function $\text{update} : D_i^{(t)} \to \bigoplus_{D_i^{(t)}} (x_j^{(t)} \otimes a_{ij})$.

The algorithm converges to the optimal solution when no component of $x^{(t)}$ changes. For single-source shortest distances, the algorithm works on the semiring $(\mathbb{R} \cup \infty, \min, +)$. For all vertices but the source, the initial distance is set to ∞. At each iteration, the algorithm tries to find a shorter distance by extending the current paths by one edge, using relaxation operations. The obtained algorithm is the well known Bellman-Ford algorithm. For single-source reachability, the semiring used is $(\{0, 1\}, \wedge, \vee)$. A dependency parameter a_{ij} is 1, if i is reachable from j and 0 otherwise. The set of reachable vertices can then be found using boolean relaxation operations. Finally, in single-source maximum reliability paths, the goal is to find the most reliable paths from a given source vertex to all other vertices in the graph. A dependency parameter a_{ij} represents the probability of reaching vertex j from vertex i. The semiring in use is $([0,1], \max, \cdot)$.

To compensate failures in the distributed execution of the Generalized Jacobi algorithm, we can simply re-initialize the components lost due to failure to ϵ, the identity element of $\oplus$. Bellman-Ford, for example, converges to the optimal solution from any start vector $x^{(0)}$ which is element-wise greater than x^* [4]:

$$\text{compensate} : x_i^{(t)} \rightarrow \begin{cases} e & \text{if } i \text{ is the source vertex} \\ \epsilon & \text{if } i \text{ belongs to a failed partition} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$

5.3 Low-Rank Matrix Factorization

A popular technique to analyze interactions between two types of entities is low-rank matrix factorization. Problems in this field typically incorporate dyadic data, for example user-story interactions in news personalization [11] or the ratings of users towards products in collaborative filtering [23]. The idea is to approximately factor a sparse $m \times n$ matrix M into the product of two matrices R and C, such that $M \approx RC$. The $m \times k$ matrix R models the latent features of the entities represented by the rows of M (e.g., users), while the $k \times n$ matrix R models the latent features of the entities represented by the columns of M (e.g., news stories or products). The strength of the relation between two different entities (e.g., the preference of a user towards a product) can be computed by the dot product $r_i^\top c_j$ between their corresponding feature vectors r_i from R and c_j from C in the low dimensional feature space.

Recently developed parallel algorithms for low-rank matrix factorization, including Distributed Stochastic Gradient Descent [16] and variations of Alternating Least Squares (ALS) [34] can be leveraged to distribute the factorization. Due to their simplicity and popularity, we focus on algorithms using the ALS technique. The goal is to train a factorization that minimizes the empirical squared error $(m_{ij} - r_i^\top c_j)^2$ for each observed interaction between a row entity i and a column entity j. This error is computed by comparing the strength of each known interaction m_{ij} between two entities i and j with the strength $r_i^\top c_j$ predicted by the factorization.

In order to find a factorization, ALS repeatedly keeps one of the unknown matrices fixed, so that the other one can be optimally recomputed. That means for example, that r_i, the i-th row of R, can be recomputed by solving a least squares problem including the i-th row of M and all the columns c_j of C that correspond to non-zero entries in the i-th row of M (cf. Figure 7). ALS then rotates between recomputing the rows of R in one step and the columns of C in the subsequent step until the training error converges.

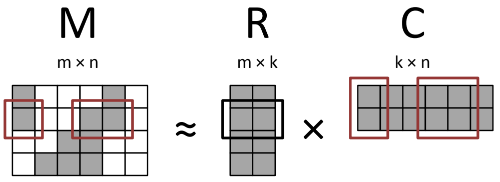

Figure 7: Dependencies for recomputing a row of R.

In the case of a failure, we lose rows of R, columns of C and parts of M. We compensate as follows: as we can always reread the lost parts of M from stable storage, we approximately recompute lost rows of R and columns of C by solving the least squares problems using the remaining feature vectors (ignoring the lost ones). If all necessary feature vectors for a row of R or a column of C are lost, we randomly re-initialize it:

$$\text{compensate} : x_i^{(t)} \rightarrow \begin{cases} \text{random vector} \in [0,1]^k & \text{if } i \text{ is lost} \\ x_i^{(t)} & \text{otherwise} \end{cases}$$

Matrix factorization with ALS is a non-convex problem [16], and our compensation function represents a jump in the parameter space. While we do not formally prove that after compensation the algorithm arrives at an equally good local minimum, we empirically validate our approach in Section 6.3 to show its applicability to real-world data.

6. EVALUATION

To evaluate the benefit of our proposed recovery mechanism, we run experiments on large datasets and simulate failures. The experimental setup for our evaluation is the following: the cluster consists of 26 machines running Java 7, Hadoop's distributed filesystem (HDFS) 1.0.4 and a customized version of Stratosphere. Each machine has two 4-core Opteron CPUs, 32 GB memory and four 1 TB disk drives.

We also run experiments with Apache Giraph [1], an open-source implementation of Pregel [25]. The findings from these experiments mirror those from the Stratosphere. We omit these experiments for lack of space.

We use three publicy available datasets for our experiments: a webgraph called Webbase[3] [5], which consists of 1,019,903,190 links between 115,657,290 webpages, a snapshot of Twitter's social network[4] [10], which contains 1,963,263,821 follower links between 51,217,936 users and a dataset of 717,872,016 ratings that 1,823,179 users gave to 136,736 songs in the Yahoo! Music community[5].

6.1 Failure-free Performance

We compare our proposed optimistic approach to a pessimistic strategy that writes distributed checkpoints. When checkpointing is enabled, the checkpoints are written to HDFS in a binary format with the default replication factor of three. Analogously to the approaches taken in Pregel and GraphLab, we checkpoint state as well as communicated dependencies [24, 25]. We run each algorithm with a degree of parallelism of 208 (one worker per core).

We look at the failure-free performance of optimistic and pessimistic recovery, in order to measure the overhead introduced by distributed checkpointing. Figure 8 shows the application of PageRank on the Webbase dataset. The x axis shows the iteration number and the y axis shows the time (in seconds) it took the system to complete the iteration. The algorithm converges after 39 iterations. The runtime with checkpointing every iteration is 79 minutes, while optimistic recovery reduces the runtime to 14 minutes. The performance of the optimistic approach is equal to the performance of running without a fault tolerance mechanism, with an iteration lasting roughly 20 seconds. We see that the pessimistic approach introduces an overhead of more than factor 5 regarding the execution time of every iteration that includes a checkpoint.

Writing checkpoints only at a certain interval improves this situation, but trades off the time it takes to write a checkpoint with the time required to repeat the iterations conducted since the last checkpoint in case of a failure. If we for example checkpoint every fifth iteration, the incurred overhead to the runtime compared to the optimal failure-free performance is still 82%, approximately 11.5

[3] http://law.di.unimi.it/webdata/webbase-2001/

[4] http://twitter.mpi-sws.org/data-icwsm2010.html

[5] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

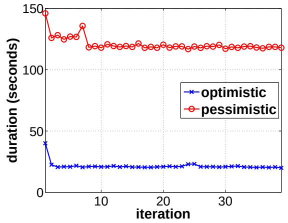

Figure 8: Failure-free performance of PageRank on the Webbase dataset.

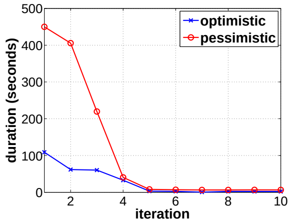

Figure 9: Failure-free performance of Connected Components on the Twitter dataset.

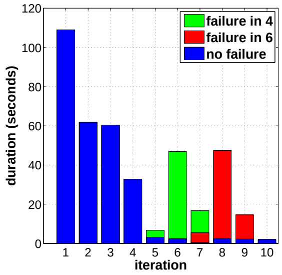

Figure 10: Recovery behavior of Connected Components on the Twitter dataset.

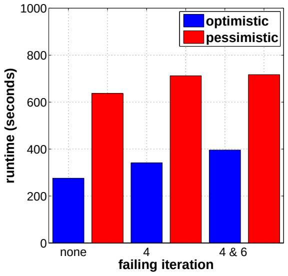

Figure 11: Runtime comparison of Connected Components on the Twitter dataset.

minutes. We see that the optimistic approach is able to outperform a pessimistic one by a factor of two to five in this case. We note that in these experiments, this was the only job running in the cluster. In busy clusters, where a lot of concurrent jobs compete for I/O and network bandwidth, the negative effect of the overhead of checkpointing might be even more dramatic.

Figure 9 shows the failure-free performance of Connected Components applied to the Twitter dataset. The checkpointing overhead varies between the iterations, due to the incremental character of the execution. With our optimistic approach, the execution takes less than 5 minutes. Activating checkpointing in every iteration increases the runtime to 19 minutes. We see that checkpointing results in a $3\times$ to $4\times$ increased runtime during the first iterations until the majority of vertices converge. Taking a checkpoint in the first iteration takes even longer than running the whole job with our optimistic scheme. After iteration 4, the majority of the state has converged, which results in a smaller workset and substantially reduces the checkpointing overhead. Again, the optimistic approach outperforms a pessimistic one that takes an early checkpoint by at least a factor of two.

Our experiments suggest that the overhead of our optimistic approach in the failure-free case (collecting few global statistics using the distributed aggregation mechanism) is virtually zero. We observe that, in the absence of failures, our prototype has the same performance as Stratosphere with checkpointing turned off, therefore we conclude that it has optimal failure-free performance.

6.2 Recovery Performance

In the following, we simulate the failure of one machine, thereby losing 8 parallel partitions of the solution. We simulate failures by dropping the parts of the state x that were assigned to the failing partitions in the failing iteration. Additionally, we discard 50% of the messages sent from the failing partitions, simulating the fact that the machine failed during the iteration and did not complete all necessary inter-machine communication. After the failing iteration, our prototype applies the compensation function and finishes the execution. We do not include the time to detect a failure and acquire a new machine into our simulations, as the incurred overhead would be the same for both a pessimistic and an optimistic approach.

To evaluate the recovery of incremental iterations, we run the Connected Components algorithm on the Twitter dataset. We simulate failures in different iterations and apply the compensation function. Figure 10 shows the overall number of iterations as well as their duration. The figure illustrates that Connected Components is very robust against failures when paired with our compensation function. Single failures in iterations 4 and 6 do not produce additional iterations, but only cause an increase in the runtime of the after next iteration by 45 seconds (which amounts to an increase of 15% in the overall runtime). This happens because the system reactivates the neighbors of failed vertices in the iteration after the failure and by this, triggers the recomputation of the failed vertices in the after next iteration. When we simulate multiple failures of different machines during one run, we observe that this process simply happens twice during the execution: The compensation of failures in iterations 4 and 6 or 5 and 8 during on run produces an overhead of approximately 35% compared to the failure-free execution. To investigate the effects of a drastic failure, we simulate a simultaneous failure of 5 machines in iteration 4 and repeat this for iteration 6. We again observe fast recovery: in both cases the overhead is less than 30% compared to the failure-free performance.

In all experiments, the additional work caused by the optimistic recovery amounts to small a fraction of the cost of writing a single checkpoint in an early iteration. The execution with a compensated single machine failure takes at most 65 seconds longer than the failure-free run, while checkpointing a single early iteration alone induces an overhead of two to five minutes. Additional to writing the checkpoints, a pessimistic approach with a checkpointing interval would have to repeat all the iterations since the last checkpoint. Figure 11 illustrates the runtimes of a pessimistic approach (with a checkpoint interval of 2 iterations) to our optimistic approach. The times for the pessimistic approach are composed of the average execution time, the average time for writing checkpoints and the execution time for iterations that need to be repeated. The figure lists the runtime of both approaches for executions with no failures, a single failure in iteration 4 and multiple failures during one run in iterations 4 and 6. Figure 11 shows that our optimistic approach is more than twice as fast in the failure-free case and at the same time provides faster recovery than a pessimistic approach in all cases.

The robustness in Connected Components is due to the sparse computational dependencies of the problem. Every minimum label propagates through its component of the graph. As social networks typically have a short average distance between vertices, the majority of the vertices find their minimum label relatively early and converge. After failure compensation in a later iteration, most vertices have a non-failed neighbor that has already found the minimum label, which they immediately receive and converge.

In order to evaluate the recovery of bulk iterations, we run PageRank on the Webbase dataset until convergence. We simulate failures in different iterations of PageRank and measure the number of iterations it takes the optimistic recovery mechanism to converge afterwards. Figure 12 shows the convergence behavior for the simulated failures in different early iterations. The x axis shows the iteration number. The y axis shows the L1-norm of the difference of the current estimate $x^{(t)}$ of the PageRank vector in iteration t to the PageRank vector $x^{(t-1)}$ in the previous iteration $t-1$ in logarithmic scale. We notice that the optimistic recovery is able to handle failures in early iterations such as the third, tenth, or fifteenth iteration extremely well with the compensation resulting only in

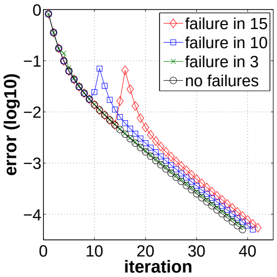

Figure 12: Recovery convergence of PageRank on the Webbase dataset.

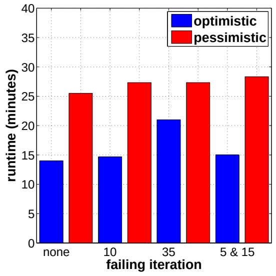

Figure 13: Runtime comparison of PageRank on the Webbase dataset.

maximum 3 additional iterations (as can be seen in the right bottom corner of Figure 12). In additional experiments, we observe the same behavior for multiple failures during one run in early iterations: failures in iterations 2 and 10 or 5 and 15 also only result in at most 3 additional iterations. Next, we simulate a simultaneous failure of five machines to investigate the effect of drastic failures: such a failure costs no overhead when it happens in iteration 3 and results in only 6 more iterations when it happens in iteration 10.

When we simulate failures in later iterations, we note that they cause more overhead: a failure in iteration 25 needs 12 more iterations to reach convergence, while a failure in iteration 35 triggers 22 additional iterations compared to the failure-free case. This behavior can be explained as follows: a compensation can be thought of as a random jump in the space of possible intermediate solutions. In later iterations, the algorithm is closer to the fixpoint, hence the random jump increases the distance to the fixpoint with a higher probability than in early iterations where the algorithm is far from the fixpoint. For a failure in iteration 35, executing these additional 22 iterations would incur an overhead of approximately eight minutes. Compared to a pessimistic approach with a checkpoint interval of five, the overhead is still less, as the pessimistic approach would have to restart from the checkpointed state of iteration 30 and again write a total of 7 checkpoints, amounting to a total overhead of more than 12 minutes. Figure 13 summarizes our findings about PageRank and compares the runtimes of our optimistic scheme to such a pessimistic approach with a checkpoint interval of five. The times for the pessimistic approach comprise the average execution time, the average time for writing checkpoints and the execution time for iterations that need to be repeated. We see that the optimistic approach outperforms the pessimistic one by nearly a factor of two in the failure-free case and for all cases, its overall runtime is shorter in light of failures.

For failures in later iterations, our findings suggest that hybrid approaches which use the optimistic approach for early iterations and switch to a pessimistic strategy later are needed. The decision when to switch could be made by observing the slope of the convergence rate. Once it starts flattening, the algorithm comes closer to the fixpoint and it would be beneficial to switch to a checkpoint-based recovery strategy. We plan to investigate this as part of our future work.

6.3 Empirical Validation of Compensability

We did not provide a formal proof for the compensability of Alternating Least Squares for low-rank matrix factorization discussed in Section 5.3. However, we can empirically validate that our optimistic approach is applicable to this algorithm. We use the Yahoo Songs dataset for this. To show the compensability of Alternating Least Squares, we implement a popular variant of the algorithm aimed at handling ratings [34] as data flow program (cf., Figure 14).

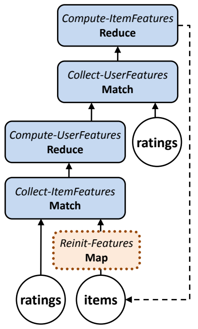

Figure 14: ALS as fixpoint algorithm in Stratosphere.

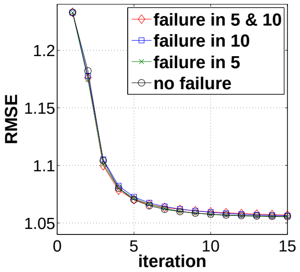

Figure 15: Optimistic recovery of failures in ALS on the Yahoo Songs dataset.

Next, we simulate failures while computing a factorization of rank 10. We simulate a single failure in iteration 5, a single failure in iteration 10 and finally have a run with multiple failures where both failures happen after another. We measure the training error (by the root mean squared error, RMSE, shown in Figure 15) of the factorization over 15 iterations. Our results show that failures result in a short increase of the error, yet until iteration 15, the error again equals that of the failure-free case. These results indicate that the ALS algorithm paired with the provided compensation strategy described in Section 5.3 is very robust to all failures, which only cause a short-lived distortion in the convergence behavior. We also repeat this experiment for the Movielens[6] dataset, where the results mirror our findings.

7. RELATED WORK

Executing iterative algorithms efficiently in parallel has received a lot of attention recently, resulting in graph-based systems [24,25], and in the integration of iterations into data flow systems [7,14,26,27,31]. The work proposed in this paper is applicable to all systems that follow a data flow or a vertex-centric programming paradigm.

While the robust characteristics of fixpoint algorithms have been known for decades, we are not aware of an approach that leverages these characteristics for the recovery of distributed, data-parallel execution of such algorithms. Rather, most distributed data processing systems [12,25], distributed storage systems [17], and recently real-time analytical systems [33] use pessimistic approaches based on periodic checkpointing and replay to recover lost state.

Systems such as *Spark* [31] offer recovery by recomputing lost partitions based on their lineage. The complex data dependencies of fixpoint algorithms however may require a full recomputation of the algorithm state. While the authors propose to use efficient in-memory checkpoints in that case, such an approach still increases the resource usage of the cluster, as several copies of the data to checkpoint have to be held in memory. During execution such a recovery strategy competes with the actual algorithm in terms of memory and network bandwidth.

Confined Recovery [25] in Pregel limits recovery to the partitions lost in a failure. The state of lost partitions is recalculated from the logs of outgoing messages of all non-failed machines in case of a failure. Confined recovery is still a pessimistic approach, which requires an increase in the amount of checkpointed data, as all outgoing messages of the system have to be logged.

Similar to our compensation function, user-defined functions to enable optimistic recovery have been proposed for long-lived trans-

[6] http://www.grouplens.org/node/73

actions. The *ConTract Model* is a mechanism for handling long-lived computations in a database context [30]. *Sagas* describe the concept of breaking long lived-transactions into a collection of sub-transactions [15]. In such systems, user-defined compensation actions are triggered in response to violations of invariants or failures of nested sub-transations during execution.

8. CONCLUSIONS AND OUTLOOK

We present a novel optimistic recovery mechanism for distributed iterative data processing using a general fixpoint programming model. Our approach eliminates the need to checkpoint to stable storage. In case of a failure, we leverage a user-defined compensation function to algorithmically bring the intermediary state of the iterative algorithm back into a form from which the algorithm still converges to the correct solution. Furthermore, we discuss how to integrate the proposed recovery mechanism into a data flow system.

We model three wide classes of problems (link analysis and centrality in networks, path enumeration in graphs, and low-rank matrix factorization) as fixpoint algorithms and describe generic compensation functions that in many cases provably converge.

Finally, we present empirical evidence that shows that our proposed optimistic approach provides optimal failure-free performance (virtually zero overhead in absence of failures). At the same time, it provides faster recovery in the majority of cases. For incrementally iterative algorithms, the recovery overhead of our approach is less than a fifth of the time it takes a pessimistic approach to only checkpoint an early intermediate state of the computation. For recovery of early iterations of bulk iterative algorithms, the induced overhead to the runtime is less than 10% and again our approach outperforms a pessimistic one in all evaluated cases.

In future work, we plan to investigate whether the compensation function can be automatically derived from invariants present in the algorithm definition. We would also like to find compensation functions for an even broader class of problems. Our experiments suggest that optimistic recovery is less effective for bulk iterative algorithms when the algorithm is already close to the fixpoint. Therefore, we plan to explore the benefits of a hybrid approach that uses optimistic recovery in early stages of the execution and switches to pessimistic recovery later. Finally, we would like to leverage memory-efficient probabilistic data structures for storing statistics to create better compensation functions.

Acknowledgments

We thank Jeffrey Ullman, Jörg Fliege, Jérôme Kunegis and Max Heimel for fruitful discussions. This research is funded by the German Research Foundation (DFG) under grant FOR 1036 'Stratosphere' and the European Union (EU) under grant no. 257859 'ROBUST'. We used data provided by 'Yahoo! Academic Relations'.

9. REFERENCES

[1] Apache Giraph, http://giraph.apache.org.

[2] Apache Hadoop, http://hadoop.apache.org.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *SoCC*, pp. 119–130, 2010.

[4] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation*. Athena Scientific, 1997.

[5] P. Boldi and S. Vigna. The Webgraph Framework I: Compression techniques. In *WWW*, pp. 595–602, 2004.

[6] P. Bonacich. Power and Centrality: A family of measures. *American Journal of Sociology*, pp. 1170–1182, 1987.

[7] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pp. 1151–1162, 2011.

[8] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, pp. 335–350, 2006.

[10] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *ICWSM*, 2010.

[11] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *WWW*, pp. 271–280, 2007.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[14] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[15] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, pp. 249–259, 1987.

[16] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pp. 69–77, 2011.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pp. 29–43, 2003.

[18] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[19] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings - New Models and Algorithms*. Springer, 2008.

[20] U. Kang, S. Papadimitriou, J. Sun, and H. Tong. Centralities in large networks: Algorithms and observations. In *SDM*, pp. 119–130, 2011.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pp. 229–238, 2009.

[22] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[23] Y. Koren, R. M. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 2009.

[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD* pp. 135–146, 2010.

[26] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[27] S. R. Mihaylov, Z. G. Ives, and S. Guha. Rex: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

[28] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E*, 74:036104, 2006.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford InfoLab*, 1999.

[30] A. Reuter and F. Schwenkreis. ConTracts - A low-level mechanism for building general-purpose workflow management-systems. *IEEE Data Eng. Bull.*, 18(1):4–10, 1995.

[31] M. Zaharia, M. Chowdhury, T. Das, D. Ankur, M. McCauley, M. Franklin, S. Shenker and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NDSI*, pp. 2–2 2012.

[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[33] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCLoud*, 2012.

[34] Y. Zhou, D. M. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pp. 337–348, 2008.